实验报告

实验目的

1. 上下文无关文法左递归消去算法的实现
    1. 理解上下文无关文法中左递归的概念及其对语法分析的影响。

    2. 掌握消去上下文无关文法中直接和间接左递归的算法。
    3. 培养运用编程语言实现文法变换的能力。
2. 文法左公共因子提取方法及实现
    1. 理解上下文无关文法中的左公共因子的概念及其对语法分析的影响。
    2. 掌握从上下文无关文法中提取左公共因子的算法，形成无二义性的语法结构。
    3. 熟练运用数据结构（如 Trie 树）处理和优化文法。
3. 上下文无关文法 FIRST 集和 FOLLOW 集求解及实现
    1. 理解上下文无关文法中 FIRST 集和 FOLLOW 集的概念及其在语法分析中的重要性。
    2. 掌握计算文法中 FIRST 集和 FOLLOW 集的算法及其实现。
    3. 培养分析和解决文法问题的能力。
4. LL(1)文法判定与预测分析器设计及实现
    1. 理解 LL(1)文法的概念及其在语法分析中的应用。
    2. 掌握判定文法是否为 LL(1)的方法。
    3. 学习设计和实现 LL(1)预测分析器的过程。
    4. 培养运用编程语言实现自顶向下语法分析的能力。

实验内容

1. 实现消去上下文无关文法中所有左递归的算法
    1. 对非终结符集合进行排序。
    2. 按顺序遍历每个非终结符，检查其候选式是否以排在其前面的非终结符开头，并进行代换。
    3. 消去直接左递归。
2. 实现从上下文无关文法中提取左公共因子的算法
    1. 对每个非终结符的候选式，识别最长的公共前缀。
    2. 构建字典树（Trie），辅助提取最长公共前缀，将公共前缀提取为新非终结符的候选式。
    3. 输出去除左公共因子的等价文法。
3. 实现求解上下文无关文法的 FIRST 集和 FOLLOW 集的算法
    1. 输入上下文无关文法。
    2. 计算每个非终结符的 FIRST 集。
    3. 计算每个非终结符的 FOLLOW 集。
4. 实现 LL(1)文法的判定算法和预测分析器的设计与实现
    1. 输入上下文无关文法。
    2. 判断文法是否为 LL(1)。
    3. 构造预测分析表。
    4. 实现预测分析器，能够根据输入串进行语法分析。

数据结构设计

类`TireNode`

```
class TrieNode:
    def __init__(self, isEnd: bool = False):
        self.children: dict[str, TrieNode] = {}
        self.isEnd: bool = isEnd

    def find_child(self, symbal: str) -> "TrieNode":
        return self.children.get(symbal)
```

用途

`TrieNode` 是 Trie 树中的节点，用于存储字符和子节点的关系，支持 Trie 树的构建和查询操作。

属性

- `children: dict[str, TrieNode]`: 存储当前节点的子节点，键为符号，值为对应的 TrieNode。
- `isEnd: bool`: 标记是否为某个产生式的结束。

方法

- `__init__`: 初始化 Trie 节点。
- `find_child`: 查找当前节点的指定子节点。

类 `Trie`

```
class Trie:
    def __init__(self, rootSym: str):
        self.rootSym: str = rootSym
        self.rootNode: TrieNode = TrieNode()

    def insert(self, symbals: list[str]) -> None:
        curNode = self.rootNode
        for sym in symbals:
            childNode = curNode.find_child(sym)
            if childNode:
                curNode = childNode
            else:
                newNode = TrieNode()
                curNode.children[sym] = newNode
                curNode = newNode
        curNode.isEnd = True

    def get_prefixes(self) -> list[list[str]]:
        def get_prefix(symbal: str, node: TrieNode) -> list[str]:
            prefix = [symbal]
            curNode = node
            while len(curNode.children) == 1 and not curNode.isEnd:
                childSym, childNode = next(iter(curNode.children.items()))
                prefix.append(childSym)  # 添加到前缀
                curNode = childNode
```

```
            return prefix if len(curNode.children) > 1 else None

        prefixes: list[list[str]] = []
        for childSym, childNode in self.rootNode.children.items():
            prefix = get_prefix(childSym, childNode)
            if prefix:
                prefixes.append(prefix)

        return prefixes

    def display(self) -> None:
        def display_help(symbal: str, node: TrieNode, prefix: str, last: bool) ->
None:
            print(prefix + "-" + symbal)
            newPrefix = prefix if not last else prefix[:-1] + " "
            newPrefix += len(symbal) * " " + "|"
            childPeers = len(node.children)
            children = iter(node.children.items())
            for i in range(childPeers):
                childSym, childNode = next(children)
                display_help(childSym, childNode, newPrefix, i == childPeers - 1)

        print(self.rootSym)
        prefix = (len(self.rootSym) - 1) * " " + "|"
        peers = len(self.rootNode.children)
        children = iter(self.rootNode.children.items())
        for i in range(peers):
            childSym, childNode = next(children)
            display_help(childSym, childNode, prefix, i == peers - 1)
```

用途

类 `Trie` 用于辅助从产生式中提取左公共因子。它通过构建一个 `Trie` 树来存储和识别产生式中的公共前缀。

属性

`rootSym: str`: Trie 树的根符号，一般用作标识。
`rootNode: TrieNode`: Trie 树的根节点。

方法

`__init__`: 初始化 Trie 树。
`insert`: 将一个产生式插入 Trie 树。
`get_prefixes`: 提取所有产生式的最长公共前缀。
`display`: 打印 Trie 树的结构。

类 CFG

```
class CFG:
    def __init__(self, read=False):
        self.terminalSyms: set[str] = set()  # 终结符号集
        self.startSym: str = None  # 开始符号
        self.grammar: dict[str, list[list[str]]] = {}  # 产生式
        self.firstSets: dict[str, set[str]] = {}  # FIRST 集
        self.followSets: dict[str, set[str]] = {}  # FOLLOW 集
        self.predictiveTable: dict[str, dict[str, list[list[str]]]] = {}  # 预测分
析表

        if read:
            self.read_grammar()

    def read_grammar(self) -> None:
        """读取文法"""
        symbal: set[str] = set()
        while True:
            try:
                self.startSym = input("请输入文法开始符号: ").strip()  # 获取开始符号
                assert len(self.startSym) == 1
                break
            except AssertionError:
                print("\033[31m上下文无关语法有且仅有一个开始符号\033[0m")
        symbal.add(self.startSym)
        print(f"\033[32m已读取开始符号: '{self.startSym}'\033[0m")

        print("请输入文法（使用 'END' 来结束输入）: ")
        while True:
            try:
                line = input()
            except EOFError:
                break

            if line.strip().upper() == "END":
                break
            if "->" in line:
                non_terminal, productions = line.split("->")
                non_terminal = non_terminal.strip()
                try:
                    assert len(self.startSym) == 1
                except AssertionError:
                    print("\033[31m上下文无关语法产生式左侧只能有一个非总结符号\033[0m")
                productions = [prod.split() for prod in productions.split("|")]
                print(
                    f"\033[32m已读取产生式: {non_terminal} -> {" | ".join([" 
".join(prod) for prod in productions])}\033[0m"
                )
                self.add_rule(non_terminal, productions)

    def set_start(self, startSym: str) -> None:
        """设置文法开始符号"""
        self.startSym = startSym
```

```
    def add_rule(self, nonterminalSym: str, productions: list[list[str]]) ->
None:
        """添加产生式规则，根据规则判断终结符号和非终结符号"""
        if nonterminalSym not in self.grammar:
            self.grammar[nonterminalSym] = productions
        else:
            self.grammar[nonterminalSym] += productions

        if nonterminalSym in self.terminalSyms:
            self.terminalSyms.remove(nonterminalSym)
        self.terminalSyms.update(
            sym
            for production in productions
            for sym in production
            if sym not in self.grammar
        )

    def eliminate_left_recursion(self) -> None:
        """消除左递归"""
        # 非终结符号集
        nonterminalSyms = list(self.grammar.keys())

        for i in range(len(nonterminalSyms)):
            nonterminalSym = nonterminalSyms[i]
            productions = self.grammar[nonterminalSym]

            # 非递归右部
            nonrecursiveProductions: list[list[str]] = []
            # 递归右部
            recursiveProductions: list[list[str]] = []

            # 生成间接右部
            for j in range(i):
                for prod in productions.copy():
                    if prod[0] == nonterminalSyms[j]:  # 间接右部生成
                        productions.remove(prod)
                        productions.extend(
                            [
                                prodj + prod[1:]
                                for prodj in self.grammar[nonterminalSyms[j]]
                            ]
                        )

            for newProd in productions:
                if newProd[0] == nonterminalSym:
                    recursiveProductions.append(newProd[1:])
                else:
                    nonrecursiveProductions.append(newProd)

            # 处理左递归
            if recursiveProductions:
                # 新非终结符号
                newNonterminalSym = f"{nonterminalSym}'"
                for prod in nonrecursiveProductions:
                    prod.append(newNonterminalSym)
                self.grammar[nonterminalSym] = nonrecursiveProductions
```

```
                for prod in recursiveProductions:
                    prod.append(newNonterminalSym)
                recursiveProductions.append(["ε"])
                self.add_rule(newNonterminalSym, recursiveProductions)
            else:
                self.grammar[nonterminalSym] = nonrecursiveProductions

    def extract_left_common_factors(self):
        """提取左公因式"""
        newGrammar: dict[str, list[list[str]]] = {key: [] for key in
self.grammar}

        for nonterminalSym, productions in self.grammar.items():
            # 如果产生式右部符号个数≤1，直接赋值
            if len(productions) <= 1:
                newGrammar[nonterminalSym] = productions
                continue

            # 将产生式插入到 Trie 中
            trie = Trie(nonterminalSym)
            for prod in productions:
                trie.insert(prod)

            # 获取最长公共因子式
            commonPrefixes = trie.get_prefixes()

            if commonPrefixes:
                newNonterminalSym = nonterminalSym
                prefixMap: list[tuple[list[str], str]] = []
                for commonPrefix in commonPrefixes:
                    newNonterminalSym = f"{newNonterminalSym}'"
                    prefixMap.append((commonPrefix, newNonterminalSym))
                    newGrammar[newNonterminalSym] = []
                    newGrammar[nonterminalSym].append(
                        commonPrefix + [newNonterminalSym]
                    )

                for prod in productions:
                    for commonPrefix, newNonterminalSym in prefixMap:
                        commonPrefixLen = len(commonPrefix)
                        if prod[:commonPrefixLen] == commonPrefix:

 newGrammar[newNonterminalSym].append(prod[commonPrefixLen:])
                            break
                    else:
                        newGrammar[nonterminalSym].append(prod)
            else:
                newGrammar[nonterminalSym] = productions

        self.grammar = newGrammar

    def compute_first(self, symbol: str) -> set[str]:
        if not self.firstSets:
            self.compute_firstSets()

        if symbol not in self.firstSets:
```

```
            return {symbol}
        if self.firstSets[symbol]:
            return self.firstSets[symbol]
        for prod in self.grammar[symbol]:
            count = 0
            for prodSym in prod:
                symFirst = self.compute_first(prodSym)
                if "ϵ" not in symFirst:
                    self.firstSets[symbol].update(symFirst)
                    break
                count += 1
                self.firstSets[symbol].update(symFirst - {"ϵ"})
            if count == len(prod):
                self.firstSets[symbol].add("ϵ")

        return self.firstSets[symbol]

    def compute_first_of_production(self, production: list[str]) -> set[str]:
        firstSet: set[str] = set()
        for symbol in production:
            first = self.compute_first(symbol)
            if "ϵ" not in first:
                firstSet.update(first)
                break
            elif symbol == production[-1]:
                firstSet.update(first)
            else:
                firstSet.update(first - {"ϵ"})

        return firstSet

    def compute_firstSets(self) -> dict[str, set[str]]:
        """计算 FIRST 集"""
        if self.firstSets:
            return self.firstSets

        # 初始化所有非终结符的 FIRST 集
        for nonterminalSym in self.grammar.keys():
            self.firstSets[nonterminalSym] = set()

        for nonterminalSym in self.firstSets.keys():
            self.compute_first(nonterminalSym)

        return self.firstSets

    def compute_follow(self, symbol: str) -> set[str]:
        if not self.followSets:
            self.compute_followSets()
        return self.followSets[symbol]

    def compute_followSets(self) -> dict[str, set[str]]:
        """计算 FOLLOW 集"""
        if self.followSets:
            return self.followSets

        # 初始化所有非终结符的 FOLLOW 集
```

```
        for nonterminal in self.grammar.keys():
            self.followSets[nonterminal] = set()
        self.followSets[self.startSym].add("$")  # $ 为输入的结束符

        # 迭代直到所有 FOLLOW 集不再变化
        changed = True
        while changed:
            changed = False
            for nonterminal, productions in self.grammar.items():
                for production in productions:
                    for i, symbol in enumerate(production):
                        if symbol not in self.terminalSyms:  # 当前符号是非终结符
                            originalSize = len(self.followSets[symbol])

                            if i + 1 < len(production):  # 右侧还有符号
                                firstSet = self.compute_first_of_production(
                                    production[i + 1 :]
                                )
                                self.followSets[symbol].update(firstSet - {"ϵ"})
                                if "ϵ" in firstSet:
                                    self.followSets[symbol].update(
                                        self.followSets[nonterminal]
                                    )
                            else:  # 如果是最后一个符号，添加非终结符的 FOLLOW 集
                                self.followSets[symbol].update(
                                    self.followSets[nonterminal]
                                )

                            changed |= originalSize !=
len(self.followSets[symbol])

        return self.followSets

    def compute_select_of_production(
        self, nonterminalSym: str, production: list[str]
    ) -> set[str]:
        """计算某个产生式的 SELECT 集"""
        selectSet: set[str] = set()
        firstSet = self.compute_first_of_production(production)
        selectSet.update(firstSet - {"ϵ"})
        if "ϵ" in firstSet:
            selectSet.update(self.compute_follow(nonterminalSym))

        return selectSet

    def is_ll1(self) -> bool:
        """判断 LL1 文法"""
        for nonterminalSym, productions in self.grammar.items():
            # 跳过只有一个产生式的非终结符号
            if len(productions) == 1:
                continue

            selectSet: set[str] = None
            for index, production in enumerate(productions):
                newSelectSet = self.compute_select_of_production(
                    nonterminalSym, production
```

```
                )
                if index == 0:
                    selectSet = newSelectSet
                selectSet.intersection_update(newSelectSet)
            if selectSet:
                return False
        return True

    def construct_predictive_table(self) -> dict[str, dict[str,
list[list[str]]]]:
        """构造 LL(1) 预测分析表"""
        self.predictiveTable = {
            nonterminal: {terminal: [] for terminal in self.terminalSyms | {"$"}}
            for nonterminal in self.grammar.keys()
        }

        for nonterminal, productions in self.grammar.items():
            for prod in productions:
                selectSet = self.compute_select_of_production(nonterminal, prod)
                for terminal in selectSet:
                    self.predictiveTable[nonterminal][terminal].append(prod)
                if ["ϵ"] == prod and prod not in
self.predictiveTable[nonterminal]["$"]:
                    self.predictiveTable[nonterminal]["$"].append(prod)

        return self.predictiveTable

    def parse(self, inputStr: list[str]) -> bool:
        if not self.predictiveTable:
            self.construct_predictive_table()
        if not self.is_ll1():
            print("\033[31m该文法不是LL(1)文法\033[0m")
            return False

        stack: list[str] = ["$", self.startSym]
        print("初始分析栈:", stack)
        inputStr.append("$")
        while stack:
            top = stack.pop()
            curSym = inputStr[0]
            action: str = None
            if top in self.terminalSyms | {"$"}:
                if top == curSym:
                    action = f"match: '{curSym}'"
                    inputStr = inputStr[1:]
                else:
                    return False
            else:
                productions = self.predictiveTable[top][curSym]
                if productions:
                    production = productions[0]
                    for sym in reversed(production):
                        if sym != "ϵ":
                            stack.append(sym)
                    action = f"{top} -> {" ".join(production)}"
                else:
```

```
                    return False

            print(f"分析栈: {stack}, 输入串: '{" ".join(inputStr)}', 动作:
{action}")

        return True

    def display(self):
        print(f"开始符号: '{self.startSym}'")
        print(f"非终结符号集: [{", ".join(self.grammar.keys())}]")
        print(f"终结符号集: [{", ".join(self.terminalSyms)}]")
        print("产生式:")
        for nonterminalSym in self.grammar:
            productions = " | ".join(
                [" ".join(production) for production in
self.grammar[nonterminalSym]]
            )
            print(f"{nonterminalSym} -> {productions}")

        if self.firstSets:
            print("FIRST 集:")
            for nonterminal in self.firstSets:
                print(
                    f"FIRST({nonterminal}) = {{{',
'.join(self.firstSets[nonterminal])}}}"
                )

        if self.followSets:
            print("FOLLOW 集:")
            for nonterminal in self.followSets:
                print(
                    f"FOLLOW({nonterminal}) = {{{',
'.join(self.followSets[nonterminal])}}}"
                )

        if self.predictiveTable:
            print("预测分析表:")
            for nonterminal, rules in self.predictiveTable.items():
                print(f"{nonterminal}:")
                for terminal, productions in rules.items():
                    if len(productions):
                        print(f"\t{terminal}:")
                    for prod in productions:
                        print(f"\t\t{nonterminal} -> {" ".join(prod)}")
```

用途

类 `CFG`（Context-Free Grammar）用于表示和操作上下文无关文法。它提供了方法来解析文法定义、消除左递归、提取左公共因子、计算 FIRST 和 FOLLOW 集合、判断文法是否为 LL(1)，并构建预测分析表和执行预测分析。

属性

- `terminalSyms: set[str]`：存储所有的终结符号。
- `startSym: str`：表示文法的开始符号。
- `grammar: dict[str, list[list[str]]]`：以字典形式存储产生式，键为非终结符，值为产生式列表，每个产生式是符号的列表。
- `firstSets: dict[str, set[str]]`：存储每个非终结符的 FIRST 集合。
- `followSets: dict[str, set[str]]`：存储每个非终结符的 FOLLOW 集合。
- `predictiveTable: dict[str, dict[str, list[list[str]]]]`：预测分析表，外层字典键为非终结符，内层字典键为终结符或结束符 "$"，值为产生式列表。

方法

- `__init__`：初始化 CFG 实例。
- `read_grammar`：从用户输入读取并构建文法，其中用户首先输入文法的开始符号，然后输入文法产生式，每个字符串以空格相隔代表一个符号，输入 END 表示结束文法输入。
- `add_rule`：添加文法规则。
- `eliminate_left_recursion`：消除左递归。
- `extract_left_common_factors`：提取左公共因子。
- `compute_firstSets`：计算所有非终结符的 FIRST 集。
- `compute_followSets`：计算所有非终结符的 FOLLOW 集。
- `is_ll1`：判断文法是否为 LL(1) 文法。
- `construct_predictive_table`：构建预测分析表。
- `parse`：执行预测分析。
- `display`：输出文法属性。

算法实现

上下文无关文法左递归消去

```python
def eliminate_left_recursion(self) -> None:
    """消除左递归"""
    # 非终结符号集
    nonterminalSyms = list(self.grammar.keys())

    for i in range(len(nonterminalSyms)):
        nonterminalSym = nonterminalSyms[i]
        productions = self.grammar[nonterminalSym]

        # 非递归右部
        nonrecursiveProductions: list[list[str]] = []
        # 递归右部
        recursiveProductions: list[list[str]] = []

        # 生成间接右部
        for j in range(i):
```

```
            for prod in productions.copy():
                if prod[0] == nonterminalSyms[j]:  # 间接右部生成
                    productions.remove(prod)
                    productions.extend(
                        [
                            prodj + prod[1:]
                            for prodj in self.grammar[nonterminalSyms[j]]
                        ]
                    )

        for newProd in productions:
            if newProd[0] == nonterminalSym:
                recursiveProductions.append(newProd[1:])
            else:
                nonrecursiveProductions.append(newProd)

        # 处理左递归
        if recursiveProductions:
            # 新非终结符号
            newNonterminalSym = f"{nonterminalSym}'"
            for prod in nonrecursiveProductions:
                prod.append(newNonterminalSym)
            self.grammar[nonterminalSym] = nonrecursiveProductions
            for prod in recursiveProductions:
                prod.append(newNonterminalSym)
            recursiveProductions.append(["ϵ"])
            self.add_rule(newNonterminalSym, recursiveProductions)
        else:
            self.grammar[nonterminalSym] = nonrecursiveProductions
```

算法步骤

1. 初始化和定义：

- 获取所有非终结符的列表 `nonterminalSyms`。
- 对每个非终结符 `nonterminalSym`，处理其产生式集 `productions`。

2. 间接左递归处理：

- 对于当前非终结符 `nonterminalSym` 的每个产生式，检查是否以列表中先于它的某个非终结符开头（间接左递归的情况）。
- 如果产生式以另一个非终结符开头，将这个产生式替换为该非终结符所有产生式的展开，从而将间接左递归转换为直接左递归。

3. 分类产生式：

- 分别识别和分类直接左递归和非左递归的产生式。
- 非左递归产生式直接保存在 `nonrecursiveProductions`。
- 左递归产生式的右部保存在 `recursiveProductions`。

4. 消除直接左递归：

- 对于存在直接左递归的非终结符，引入一个新的非终结符 `newNonterminalSym`（例如，原非终结符为 `A`，则新非终结符为 `A'`）。

- 将所有非左递归产生式修改为在末尾添加新非终结符 `newNonterminalSym`。
- 新非终结符的产生式为：对每个原左递归产生式的右部追加新非终结符 `newNonterminalSym`，并添加一条产生空串（`ε`）的产生式以处理递归的终止。

5. 更新文法：

- 更新当前非终结符的产生式列表为修改后的非递归产生式。
- 将新的非终结符及其产生式添加到文法中。

算法测试

使用`pytest`进行单元测试：

```
import pytest
def test_cfg_eliminate_left_recursion1():
    cfg = CFG(False)

    cfg.set_start("S")
    cfg.add_rule("S", [["S", "+", "T"], ["T"]])
    cfg.add_rule("T", [["T", "*", "F"], ["F"]])
    cfg.add_rule("F", [["(", "E", ")"], ["id"]])

    print("原始文法:")
    cfg.display()

    cfg.eliminate_left_recursion()

    print("消去左递归后的文法:")
    cfg.display()

    expected_rules: dict[str, list[list[str]]] = {
        "S": [["T", "S'"]],
        "S'": [["+", "T", "S'"], ["ϵ"]],
        "T": [["F", "T'"]],
        "T'": [["*", "F", "T'"], ["ϵ"]],
        "F": [["(", "E", ")"], ["id"]],
    }

    assert sorted(cfg.grammar.keys()) == sorted(expected_rules.keys())
    for non_terminal, productions in expected_rules.items():
        assert sorted(["".join(prod) for prod in cfg.grammar[non_terminal]]) ==
sorted(
            ["".join(prod) for prod in productions]
        )


def test_cfg_eliminate_left_recursion2():
    cfg = CFG(False)

    cfg.set_start("S")
    cfg.add_rule("S", [["A", "c"], ["c"]])
    cfg.add_rule("A", [["B", "b"], ["b"]])
    cfg.add_rule("B", [["S", "a"], ["a"]])

    print("原始文法:")
```

```
    cfg.display()

    cfg.eliminate_left_recursion()

    print("消去左递归后的文法:")
    cfg.display()

    expected_rules: dict[str, list[list[str]]] = {
        "S": [["A", "c"], ["c"]],
        "A": [["B", "b"], ["b"]],
        "B": [["b", "c", "a", "B'"], ["c", "a", "B'"], ["a", "B'"]],
        "B'": [["b", "c", "a", "B'"], ["ϵ"]],
    }

    assert sorted(cfg.grammar.keys()) == sorted(expected_rules.keys())
    for non_terminal, productions in expected_rules.items():
        assert sorted(["".join(prod) for prod in cfg.grammar[non_terminal]]) ==
sorted(
            ["".join(prod) for prod in productions]
        )
```

测试结果:

```
tests/test_cfg.py::test_cfg_eliminate_left_recursion1
原始文法:
开始符号: 'S'
非终结符号集: [S, T, F]
终结符号集: [+, id, (, *, E, )]
产生式:
S -> S + T | T
T -> T * F | F
F -> ( E ) | id
消去左递归后的文法:
开始符号: 'S'
非终结符号集: [S, T, F, S', T']
终结符号集: [+, id, (, ϵ, *, E, )]
产生式:
S -> T S'
T -> F T'
F -> ( E ) | id
S' -> + T S' | ϵ
T' -> * F T' | ϵ
PASSED

tests/test_cfg.py::test_cfg_eliminate_left_recursion2
原始文法:
开始符号: 'S'
非终结符号集: [S, A, B]
终结符号集: [c, a, b]
产生式:
S -> A c | c
A -> B b | b
B -> S a | a
消去左递归后的文法:
开始符号: 'S'
```

```
非终结符号集: [S, A, B, B']
终结符号集: [ϵ, c, a, b]
产生式:
S -> A c | c
A -> B b | b
B -> a B' | c a B' | b c a B'
B' -> b c a B' | ϵ
PASSED
```

文法左公共因子提取

```python
def extract_left_common_factors(self):
    """提取左公因式"""
    newGrammar: dict[str, list[list[str]]] = {key: [] for key in self.grammar}

    for nonterminalSym, productions in self.grammar.items():
        # 如果产生式右部符号个数≤1, 直接赋值
        if len(productions) <= 1:
            newGrammar[nonterminalSym] = productions
            continue

        # 将产生式插入到 Trie 中
        trie = Trie(nonterminalSym)
        for prod in productions:
            trie.insert(prod)

        # 获取最长公共因子式
        commonPrefixes = trie.get_prefixes()

        if commonPrefixes:
            newNonterminalSym = nonterminalSym
            prefixMap: list[tuple[list[str], str]] = []
            for commonPrefix in commonPrefixes:
                newNonterminalSym = f"{newNonterminalSym}'"
                prefixMap.append((commonPrefix, newNonterminalSym))
                newGrammar[newNonterminalSym] = []
                newGrammar[nonterminalSym].append(
                    commonPrefix + [newNonterminalSym]
                )

            for prod in productions:
                for commonPrefix, newNonterminalSym in prefixMap:
                    commonPrefixLen = len(commonPrefix)
                    if prod[:commonPrefixLen] == commonPrefix:

 newGrammar[newNonterminalSym].append(prod[commonPrefixLen:])
                        break
                else:
                    newGrammar[nonterminalSym].append(prod)
        else:
            newGrammar[nonterminalSym] = productions

    self.grammar = newGrammar
```

算法步骤

1. 初始化：

- 创建一个新的文法存储结构 `newGrammar`，用于保存修改后的文法。

2. 遍历文法中的每个非终结符：

- 对于每个非终结符 `nonterminalSym` 和其对应的产生式集 `productions`，进行处理。

3. 单一产生式快速处理：

- 如果一个非终结符只有一个产生式，或者其所有产生式都不共享任何前缀，直接将其复制到 `newGrammar`，无需进一步处理。

4. 构建 `Trie` 树：

- 使用 `Trie` 数据结构来插入并存储所有产生式，以便有效地发现共享的前缀。
- 对每个产生式，将其插入到 `Trie` 树中。

5. 提取公共前缀：

- 从 `Trie` 树中获取所有最长的公共前缀 `commonPrefixes`。
- 对于每个发现的公共前缀，创建一个新的非终结符 `newNonterminalSym`（如，原非终结符为 `A`，则新非终结符为 `A'`）。

6. 更新产生式：

- 将每个公共前缀和新非终结符的组合添加到原非终结符的产生式列表中。
- 对每个原始产生式，检查它是否以某个公共前缀开始：
  - 如果是，将除去公共前缀后的剩余部分添加为新非终结符的产生式。
  - 如果不是，不修改该产生式。

7. 文法更新：

- 将处理过的新文法 `newGrammar` 替换原有的文法结构。

算法测试

使用`pytest`进行单元测试:

```
import pytest
def test_extract_left_common_factors():
    cfg = CFG(False)

    cfg.set_start("S")
    cfg.add_rule(
        "S",
        [
            list("apple"),
            list("apply"),
            list("application"),
            list("ball"),
            list("bat"),
            list("bath"),
            list("Xb"),
```

```
        ],
    )
    cfg.add_rule("X", [list("ab"), list("ac"), list("ad")])

    print("原始文法:")
    cfg.display()

    cfg.extract_left_common_factors()

    print("消去公因式后的文法:")
    cfg.display()

    expected_rules: dict[str, list[list[str]]] = {
        "S": [list("appl") + ["S'"], ["b", "a", "S''"], ["X", "b"]],
        "S'": [["e"], ["y"], list("ication")],
        "S''": [["l", "l"], ["t"], ["t", "h"]],
        "X": [["a", "X'"]],
        "X'": [["b"], ["c"], ["d"]],
    }

    assert sorted(cfg.grammar.keys()) == sorted(expected_rules.keys())
    for non_terminal, productions in expected_rules.items():
        assert sorted(["".join(prod) for prod in cfg.grammar[non_terminal]]) ==
sorted(
            ["".join(prod) for prod in productions]
        )
```

测试结果:

```
tests/test_cfg.py::test_extract_left_common_factors
原始文法:
开始符号: 'S'
非终结符号集: [S, X]
终结符号集: [h, t, a, d, y, e, b, l, c, o, n, p, i]
产生式:
S -> a p p l e | a p p l y | a p p l i c a t i o n | b a l l | b a t | b a t h |
X b
X -> a b | a c | a d
消去公因式后的文法:
开始符号: 'S'
非终结符号集: [S, X, S', S'', X']
终结符号集: [h, t, a, d, y, e, b, l, c, o, n, p, i]
产生式:
S -> a p p l S' | b a S'' | X b
X -> a X'
S' -> e | y | i c a t i o n
S'' -> l l | t | t h
X' -> b | c | d
PASSED
```

上下文无关文法 FIRST 集和 FOLLOW 集求解

```
def compute_first(self, symbol: str) -> set[str]:
    if not self.firstSets:
        self.compute_firstSets()

    if symbol not in self.firstSets:
        return {symbol}
    if self.firstSets[symbol]:
        return self.firstSets[symbol]
    for prod in self.grammar[symbol]:
        count = 0
        for prodSym in prod:
            symFirst = self.compute_first(prodSym)
            if "ϵ" not in symFirst:
                self.firstSets[symbol].update(symFirst)
                break
            count += 1
            self.firstSets[symbol].update(symFirst - {"ϵ"})
        if count == len(prod):
            self.firstSets[symbol].add("ϵ")

    return self.firstSets[symbol]

    return firstSet

def compute_firstSets(self) -> dict[str, set[str]]:
    """计算 FIRST 集"""
    if self.firstSets:
        return self.firstSets

    # 初始化所有非终结符的 FIRST 集
    for nonterminalSym in self.grammar.keys():
        self.firstSets[nonterminalSym] = set()

    for nonterminalSym in self.firstSets.keys():
        self.compute_first(nonterminalSym)

    return self.firstSets

def compute_follow(self, symbol: str) -> set[str]:
    if not self.followSets:
        self.compute_followSets()
    return self.followSets[symbol]

def compute_followSets(self) -> dict[str, set[str]]:
    """计算 FOLLOW 集"""
    if self.followSets:
        return self.followSets

    # 初始化所有非终结符的 FOLLOW 集
    for nonterminal in self.grammar.keys():
        self.followSets[nonterminal] = set()
    self.followSets[self.startSym].add("$")  # $ 为输入的结束符
```

```
    # 迭代直到所有 FOLLOW 集不再变化
    changed = True
    while changed:
        changed = False
        for nonterminal, productions in self.grammar.items():
            for production in productions:
                for i, symbol in enumerate(production):
                    if symbol not in self.terminalSyms:  # 当前符号是非终结符
                        originalSize = len(self.followSets[symbol])

                        if i + 1 < len(production):  # 右侧还有符号
                            firstSet = self.compute_first_of_production(
                                production[i + 1 :]
                            )
                            self.followSets[symbol].update(firstSet - {"ϵ"})
                            if "ϵ" in firstSet:
                                self.followSets[symbol].update(
                                    self.followSets[nonterminal]
                                )
                        else:  # 如果是最后一个符号，添加非终结符的 FOLLOW 集
                            self.followSets[symbol].update(
                                self.followSets[nonterminal]
                            )

                        changed |= originalSize != len(self.followSets[symbol])

    return self.followSets
```

算法步骤

计算 FIRST 集

1. 初始化和检查：

- 首先检查是否已经存在计算好的 `firstSets`，如果存在，直接返回该集合。
- 如果不存在，为每个非终结符初始化一个空的 FIRST 集合。

2. 递归计算 FIRST 集：

- 对每个非终结符，遍历其所有产生式。
- 对每个产生式中的每个符号进行处理：
- 调用 `compute_first` 函数递归地计算每个符号的 FIRST 集。
- 如果当前符号的 FIRST 集不包含空串 ε，则将该 FIRST 集添加到非终结符的 FIRST 集中，并停止处理当前产生式。
- 如果包含空串 ε，则添加除 ε 之外的所有符号到 FIRST 集。
- 如果所有符号的 FIRST 集均包含空串 ε，则将空串 ε 也添加到当前非终结符的 FIRST 集中。

计算 FOLLOW 集

1. 初始化 FOLLOW 集：

- 检查是否已经计算过 FOLLOW 集，如果已经计算，则直接返回。
- 否则，为每个非终结符初始化一个空的 FOLLOW 集合，并将结束符 $ 添加到开始符号的 FOLLOW 集中。

2. 迭代计算 FOLLOW 集：

- 循环遍历所有产生式，直到 FOLLOW 集不再变化。
- 对每个产生式中的每个非终结符符号进行处理：
  - 如果该符号后面还有其他符号，计算这些符号的 FIRST 集并添加到当前非终结符的 FOLLOW 集中（不包括 ε）。
  - 如果该符号后面的符号的 FIRST 集包含 ε 或该符号是产生式的最后一个符号，则将产生式左侧非终结符的 FOLLOW 集添加到该非终结符的 FOLLOW 集中。
- 在每次迭代后检查 FOLLOW 集的大小是否有变化，如果没有变化，则终止循环。

算法测试

使用`pytest`进行单元测试:

```
def test_compute_first1():
    cfg = CFG(False)

    cfg.set_start("S")
    cfg.add_rule("S", [["A", "B"]])
    cfg.add_rule("A", [["a"], ["ϵ"]])
    cfg.add_rule("B", [["b"]])

    print("文法:")
    cfg.display()

    firstSets = cfg.compute_firstSets()
    print("FIRST 集:", firstSets)

    expectedFirstSets: dict[str, set[str]] = {
        "S": {"a", "b"},
        "A": {"a", "ϵ"},
        "B": {"b"},
    }

    assert len(firstSets) == len(expectedFirstSets)
    for nonterminalSym in expectedFirstSets.keys():
        assert firstSets[nonterminalSym] == expectedFirstSets[nonterminalSym]


def test_compute_first2():
    cfg = CFG(False)

    cfg.set_start("E")
    cfg.add_rule("E", [["T", "X"]])
    cfg.add_rule("T", [["int", "Y"], ["(", "E", ")"]])
    cfg.add_rule("X", [["+", "E"], ["ϵ"]])
    cfg.add_rule("Y", [["*", "T"], ["ϵ"]])

    print("文法:")
    cfg.display()

    firstSets = cfg.compute_firstSets()
    print("FIRST 集:", firstSets)
```

```
    expectedFirstSets: dict[str, set[str]] = {
        "E": {"(", "int"},
        "T": {"(", "int"},
        "X": {"+", "ϵ"},
        "Y": {"*", "ϵ"},
    }

    assert len(firstSets) == len(expectedFirstSets)
    for nonterminalSym in expectedFirstSets.keys():
        assert firstSets[nonterminalSym] == expectedFirstSets[nonterminalSym]


def test_compute_follow1():
    cfg = CFG(False)

    cfg.set_start("S")
    cfg.add_rule("S", [["A", "B"]])
    cfg.add_rule("A", [["a"], ["ϵ"]])
    cfg.add_rule("B", [["b"]])

    print("文法:")
    cfg.display()

    followSets = cfg.compute_followSets()
    print("FOLLOW 集:", followSets)

    expectedFollowSets: dict[str, set[str]] = {
        "S": {"$"},
        "A": {"b"},
        "B": {"$"},
    }

    assert len(followSets) == len(expectedFollowSets)
    for nonterminalSym in expectedFollowSets.keys():
        assert followSets[nonterminalSym] == expectedFollowSets[nonterminalSym]


def test_compute_follow2():
    cfg = CFG(False)

    cfg.set_start("E")
    cfg.add_rule("E", [["T", "X"]])
    cfg.add_rule("T", [["int", "Y"], ["(", "E", ")"]])
    cfg.add_rule("X", [["+", "E"], ["ϵ"]])
    cfg.add_rule("Y", [["*", "T"], ["ϵ"]])

    print("文法:")
    cfg.display()

    followSets = cfg.compute_followSets()
    print("FOLLOW 集:", followSets)

    expectedFollowSets: dict[str, set[str]] = {
        "E": {"$", ")"},
        "X": {"$", ")"},
        "T": {"+", "$", ")"},
```

```
        "Y": {"+", "$", ")"},
    }

    assert len(followSets) == len(expectedFollowSets)
    for nonterminalSym in expectedFollowSets.keys():
        assert followSets[nonterminalSym] == expectedFollowSets[nonterminalSym]


def test_compute_follow3():
    cfg = CFG(False)

    cfg.set_start("E")
    cfg.add_rule("E", [["T", "E'"]])
    cfg.add_rule("E'", [["+", "T", "E'"], ["ϵ"]])
    cfg.add_rule("T", [["F", "T'"]])
    cfg.add_rule("T'", [["*", "F", "T'"], ["ϵ"]])
    cfg.add_rule("F", [["(", "E", ")"], ["id"]])

    print("文法:")
    cfg.display()

    followSets = cfg.compute_followSets()
    print("FOLLOW 集:", followSets)

    expectedFollowSets: dict[str, set[str]] = {
        "E": {"$", ")"},
        "E'": {"$", ")"},
        "T": {"+", "$", ")"},
        "T'": {"+", "$", ")"},
        "F": {"*", "+", "$", ")"},
    }

    assert len(followSets) == len(expectedFollowSets)
    for nonterminalSym in expectedFollowSets.keys():
        assert followSets[nonterminalSym] == expectedFollowSets[nonterminalSym]
```

测试结果:

```
tests/test_cfg.py::test_compute_first1 文法:
开始符号: 'S'
非终结符号集: [S, A, B]
终结符号集: [a, ϵ, b]
产生式:
S -> A B
A -> a | ϵ
B -> b
FIRST 集: {'S': {'b', 'a'}, 'A': {'ϵ', 'a'}, 'B': {'b'}}
PASSED
tests/test_cfg.py::test_compute_first2 文法:
开始符号: 'E'
非终结符号集: [E, T, X, Y]
终结符号集: [int, +, (, ϵ, ), *]
产生式:
E -> T X
T -> int Y | ( E )
```

```
X -> + E | ϵ
Y -> * T | ϵ
FIRST 集: {'E': {'int', '('}, 'T': {'int', '('}, 'X': {'ϵ', '+'}, 'Y': {'*',
'ϵ'}}
PASSED
tests/test_cfg.py::test_compute_follow1 文法:
开始符号: 'S'
非终结符号集: [S, A, B]
终结符号集: [a, ϵ, b]
产生式:
S -> A B
A -> a | ϵ
B -> b
FOLLOW 集: {'S': {'$'}, 'A': {'b'}, 'B': {'$'}}
PASSED
tests/test_cfg.py::test_compute_follow2 文法:
开始符号: 'E'
非终结符号集: [E, T, X, Y]
终结符号集: [int, +, (, ϵ, ), *]
产生式:
E -> T X
T -> int Y | ( E )
X -> + E | ϵ
Y -> * T | ϵ
FOLLOW 集: {'E': {')', '$'}, 'T': {'$', ')', '+'}, 'X': {')', '$'}, 'Y': {'$',
')', '+'}}
PASSED
tests/test_cfg.py::test_compute_follow3 文法:
开始符号: 'E'
非终结符号集: [E, E', T, T', F]
终结符号集: [id, (, *, ϵ, ), +]
产生式:
E -> T E'
E' -> + T E' | ϵ
T -> F T'
T' -> * F T' | ϵ
F -> ( E ) | id
FOLLOW 集: {'E': {')', '$'}, "E'": {')', '$'}, 'T': {'$', ')', '+'}, "T'": {'$',
')', '+'}, 'F': {'$', '*', ')', '+'}}
PASSED
```

LL(1)文法判定与预测分析器

```python
def compute_first_of_production(self, production: list[str]) -> set[str]:
    """计算某个产生式的 FIRST 集"""
    firstSet: set[str] = set()
    for symbol in production:
        first = self.compute_first(symbol)
        if "ϵ" not in first:
            firstSet.update(first)
            break
        elif symbol == production[-1]:
            firstSet.update(first)
        else:
            firstSet.update(first - {"ϵ"})
```

```
    return firstSet

def compute_select_of_production(
    self, nonterminalSym: str, production: list[str]
) -> set[str]:
    """计算某个产生式的 SELECT 集"""
    selectSet: set[str] = set()
    firstSet = self.compute_first_of_production(production)
    selectSet.update(firstSet - {"ϵ"})
    if "ϵ" in firstSet:
        selectSet.update(self.compute_follow(nonterminalSym))

    return selectSet

def is_ll1(self) -> bool:
    """判断 LL1 文法"""
    for nonterminalSym, productions in self.grammar.items():
        # 跳过只有一个产生式的非终结符号
        if len(productions) == 1:
            continue

        selectSet: set[str] = None
        for index, production in enumerate(productions):
            newSelectSet = self.compute_select_of_production(
                nonterminalSym, production
            )
            if index == 0:
                selectSet = newSelectSet
            selectSet.intersection_update(newSelectSet)
        if selectSet:
            return False
    return True

def construct_predictive_table(self) -> dict[str, dict[str, list[list[str]]]]:
    """构造 LL(1) 预测分析表"""
    self.predictiveTable = {
        nonterminal: {
            terminal: [] for terminal in self.terminalSyms - {"ϵ"} | {"$"}
        }
        for nonterminal in self.grammar.keys()
    }

    for nonterminal, productions in self.grammar.items():
        for prod in productions:
            selectSet = self.compute_select_of_production(nonterminal, prod)
            for terminal in selectSet:
                self.predictiveTable[nonterminal][terminal].append(prod)
            if ["ϵ"] == prod and prod not in self.predictiveTable[nonterminal]
["$"]:
                self.predictiveTable[nonterminal]["$"].append(prod)

    return self.predictiveTable

def parse(self, inputStr: list[str]) -> bool:
    if not self.predictiveTable:
```

```
        self.construct_predictive_table()
    if not self.is_ll1():
        print("\033[31m该文法不是LL(1)文法\033[0m")
        return False

    stack: list[str] = ["$", self.startSym]
    print("初始分析栈:", stack)
    inputStr.append("$")
    while stack:
        top = stack.pop()
        curSym = inputStr[0]
        action: str = None
        if top in self.terminalSyms | {"$"}:
            if top == curSym:
                action = f"match: '{curSym}'"
                inputStr = inputStr[1:]
            else:
                return False
        else:
            productions = self.predictiveTable[top][curSym]
            if productions:
                production = productions[0]
                for sym in reversed(production):
                    if sym != "ε":
                        stack.append(sym)
                action = f"{top} -> {" ".join(production)}"
            else:
                return False

        print(f"分析栈: {stack}, 输入串: '{" ".join(inputStr)}', 动作: {action}")

    return True
```

算法步骤

判断 LL(1) 文法算法

1. 遍历文法的所有非终结符和其产生式：

- 如果一个非终结符只有一个产生式，直接跳过，因为单一产生式不会引起选择集的冲突。

2. 计算选择集（SELECT SET）：

- 对每个产生式，计算其选择集，选择集是产生式在特定上下文中可能开始的终结符集合。
- 选择集的计算依赖于产生式的 FIRST 集和 FOLLOW 集：
  - 使用 `compute_select_of_production` 方法，该方法首先计算产生式的 FIRST 集。
  - 如果产生式的 FIRST 集包含空串（ε），则将当前非终结符的 FOLLOW 集也并入到选择集中。

3. 检查选择集的交集：

- 对于同一非终结符的所有产生式，检查它们的选择集之间是否有交集。
- 如果任何两个产生式的选择集有交集，则该文法不是 LL(1)。

构造预测分析表

1. 初始化预测分析表：

- 为每个非终结符和每个可能的输入符号（终结符和结束符 $）初始化一个空列表。

2. 填充预测分析表：

- 遍历每个非终结符的每个产生式，计算其选择集。
- 对于产生式的每个终结符在选择集中，将产生式添加到预测分析表的相应位置。
- 如果产生式可以推导出空串（ε），并且该产生式没有被包含在结束符的条目中，将其添加到结束符的条目。

语法分析

1. 初始化分析栈：

- 将结束符 $ 和开始符号压入栈中。
- 如果在分析过程中发现文法不是 LL(1)，输出错误信息并终止分析。

2. 循环处理输入串：

- 比较栈顶符号和当前输入符号：
  - 如果栈顶是终结符或结束符，并且与输入符号匹配，从栈中弹出，并移动输入指针。
  - 如果栈顶是非终结符，查找预测分析表，找到匹配的产生式并将其逆序压入栈中。
  - 如果找不到匹配的产生式或栈顶符号与输入符号不匹配，则分析失败。

3. 打印动作和状态：

- 在每一步操作中，输出当前的分析栈、输入串和执行的动作（匹配或应用产生式）。

4. 完成分析：

- 当分析栈为空且输入串完全被读取时，分析成功。

算法测试

使用`pytest`进行单元测试:

```
import pytest
def test_is_ll1_1():
    cfg = CFG(False)

    cfg.set_start("S")
    cfg.add_rule("S", [["a", "A", "S"]])
    cfg.add_rule("S", [["b"]])
    cfg.add_rule("A", [["b", "A"]])
    cfg.add_rule("A", [["ϵ"]])

    print("文法:")
    cfg.display()

    print("SELECT 集:")
    for nonterminalSym, productions in cfg.grammar.items():
        for prod in productions:
            selectSet = cfg.compute_select_of_production(nonterminalSym, prod)
            print(
```

```
                f"SELECT({nonterminalSym} -> {" ".join(prod)}) = {{{",
".join(selectSet)}}}"
            )

    assert cfg.is_ll1() == False


def test_is_ll1_2():
    cfg = CFG(False)

    cfg.set_start("E")
    cfg.add_rule("E", [["T", "E'"]])
    cfg.add_rule("E'", [["+", "T", "E'"], ["ϵ"]])
    cfg.add_rule("T", [["F", "T'"]])
    cfg.add_rule("T'", [["*", "F", "T'"], ["ϵ"]])
    cfg.add_rule("F", [["(", "E", ")"], ["id"]])

    print("文法:")
    cfg.display()

    print("SELECT 集:")
    for nonterminalSym, productions in cfg.grammar.items():
        for prod in productions:
            selectSet = cfg.compute_select_of_production(nonterminalSym, prod)
            print(
                f"SELECT({nonterminalSym} -> {" ".join(prod)}) = {{{",
".join(selectSet)}}}"
            )

    assert cfg.is_ll1() == True


def test_construct_predictive_table1():
    cfg = CFG(False)

    cfg.set_start("S")
    cfg.add_rule("S", [["A", "B"]])
    cfg.add_rule("A", [["a", "A"], ["ϵ"]])
    cfg.add_rule("B", [["b"]])

    print("文法:")
    cfg.display()

    print("SELECT 集:")
    for nonterminalSym, productions in cfg.grammar.items():
        for prod in productions:
            selectSet = cfg.compute_select_of_production(nonterminalSym, prod)
            print(
                f"SELECT({nonterminalSym} -> {" ".join(prod)}) = {{{",
".join(selectSet)}}}"
            )

    print("预测分析表:")
    predictiveTable = cfg.construct_predictive_table()
    print(predictiveTable)
```

```python
def test_construct_predictive_table2():
    cfg = CFG(False)

    cfg.set_start("S")
    cfg.add_rule("S", [["i", "E", "t", "S", "S'"], ["a"]])
    cfg.add_rule("S'", [["e", "S"], ["ϵ"]])
    cfg.add_rule("E", [["b"]])

    print("文法:")
    cfg.display()

    print("SELECT 集:")
    for nonterminalSym, productions in cfg.grammar.items():
        for prod in productions:
            selectSet = cfg.compute_select_of_production(nonterminalSym, prod)
            print(
                f"SELECT({nonterminalSym} -> {" ".join(prod)}) = {{{",
".join(selectSet)}}}"
            )

    print("预测分析表:")
    predictiveTable = cfg.construct_predictive_table()
    print(predictiveTable)

    assert cfg.is_ll1() == False
```

测试结果:

```
tests/test_cfg.py::test_is_ll1_1 文法:
开始符号: 'S'
非终结符号集: [S, A]
终结符号集: [ϵ, b, a]
产生式:
S -> a A S | b
A -> b A | ϵ
SELECT 集:
SELECT(S -> a A S) = {a}
SELECT(S -> b) = {b}
SELECT(A -> b A) = {b}
SELECT(A -> ϵ) = {b, a}
PASSED
tests/test_cfg.py::test_is_ll1_2 文法:
开始符号: 'E'
非终结符号集: [E, E', T, T', F]
终结符号集: [id, (, *, ϵ, ), +]
产生式:
E -> T E'
E' -> + T E' | ϵ
T -> F T'
T' -> * F T' | ϵ
F -> ( E ) | id
SELECT 集:
SELECT(E -> T E') = {id, (}
SELECT(E' -> + T E') = {+}
```

```
SELECT(E' -> ϵ) = {), $}
SELECT(T -> F T') = {id, (}
SELECT(T' -> * F T') = {*}
SELECT(T' -> ϵ) = {+, ), $}
SELECT(F -> ( E )) = {(}
SELECT(F -> id) = {id}
PASSED
tests/test_cfg.py::test_construct_predictive_table1 文法:
开始符号: 'S'
非终结符号集: [S, A, B]
终结符号集: [a, ϵ, b]
产生式:
S -> A B
A -> a A | ϵ
B -> b
SELECT 集:
SELECT(S -> A B) = {b, a}
SELECT(A -> a A) = {a}
SELECT(A -> ϵ) = {b}
SELECT(B -> b) = {b}
预测分析表:
{'S': {'ϵ': [], 'b': [['A', 'B']], '$': [], 'a': [['A', 'B']]}, 'A': {'ϵ': [],
'b': [['ϵ']], '$': [['ϵ']], 'a': [['a', 'A']]}, 'B': {'ϵ': [], 'b': [['b']], '$':
[], 'a': []}}
PASSED
tests/test_cfg.py::test_construct_predictive_table2 文法:
开始符号: 'S'
非终结符号集: [S, S', E]
终结符号集: [i, a, b, ϵ, t, e]
产生式:
S -> i E t S S' | a
S' -> e S | ϵ
E -> b
SELECT 集:
SELECT(S -> i E t S S') = {i}
SELECT(S -> a) = {a}
SELECT(S' -> e S) = {e}
SELECT(S' -> ϵ) = {$, e}
SELECT(E -> b) = {b}
预测分析表:
{'S': {'i': [['i', 'E', 't', 'S', "S'"]], 'b': [], 'e': [], '$': [], 'ϵ': [],
't': [], 'a': [['a']]}, "S'": {'i': [], 'b': [], 'e': [['e', 'S'], ['ϵ']], '$':
[['ϵ']], 'ϵ': [], 't': [], 'a': []}, 'E': {'i': [], 'b': [['b']], 'e': [], '$':
[], 'ϵ': [], 't': [], 'a': []}}
PASSED
```

测试用例与结果分析

```python
from lab3 import *

# 测试程序
def main():
    # 从输入读取文法
    cfg = CFG(read=True)
```

```
    print("\n原始文法:")
    cfg.display()

    cfg.eliminate_left_recursion()
    cfg.extract_left_common_factors()

    print("文法是否满足LL(1):", cfg.is_ll1())
    cfg.construct_predictive_table()

    print("\n处理后的文法:")
    cfg.display()

    while True:
        try:
            inputStr = input("请输入待分析串:").strip()
            if inputStr:
                print(f"\033[32m已获取输入串: '{inputStr}'\033[0m")
                print(f"分析结果: {cfg.parse(inputStr.split(" "))}")
        except EOFError:
            break

    print("程序结束")

if __name__ == "__main__":
    main()
```

文法

```
S
S -> S + T | T
T -> T * F | F
F -> ( E ) | id
END
```

测试字符串列表

```
id + id * id
```

结果

```
poetry run python main.py < cases/input1.txt
请输入文法开始符号: 已读取开始符号: 'S'
请输入文法（使用 'END' 来结束输入）:
已读取产生式: S -> S + T | T
已读取产生式: T -> T * F | F
已读取产生式: F -> ( E ) | id

原始文法:
开始符号: 'S'
非终结符号集: [S, T, F]
终结符号集: [E, *, id, ), +, (]
产生式:
S -> S + T | T
```

```
T -> T * F | F
F -> ( E ) | id
文法是否满足LL(1): True

处理后的文法:
开始符号: 'S'
非终结符号集: [S, T, F, S', T']
终结符号集: [E, *, id, ϵ, ), +, (]
产生式:
S -> T S'
T -> F T'
F -> ( E ) | id
S' -> + T S' | ϵ
T' -> * F T' | ϵ
FIRST 集:
FIRST(S) = {id, (}
FIRST(T) = {id, (}
FIRST(F) = {id, (}
FIRST(S') = {ϵ, +}
FIRST(T') = {ϵ, *}
FOLLOW 集:
FOLLOW(S) = {$}
FOLLOW(T) = {$, +}
FOLLOW(F) = {*, $, +}
FOLLOW(S') = {$}
FOLLOW(T') = {$, +}
预测分析表:
S:
        id:
                S -> T S'
        (:
                S -> T S'
T:
        id:
                T -> F T'
        (:
                T -> F T'
F:
        id:
                F -> id
        (:
                F -> ( E )
S':
        $:
                S' -> ϵ
        +:
                S' -> + T S'
T':
        *:
                T' -> * F T'
        $:
                T' -> ϵ
        +:
                T' -> ϵ
请输入待分析串:请输入待分析串:已获取输入串: 'id + id * id'
初始分析栈: ['$', 'S']
```

```
分析栈: ['$', "S'", 'T'], 输入串: 'id + id * id $', 动作: S -> T S'
分析栈: ['$', "S'", "T'", 'F'], 输入串: 'id + id * id $', 动作: T -> F T'
分析栈: ['$', "S'", "T'", 'id'], 输入串: 'id + id * id $', 动作: F -> id
分析栈: ['$', "S'", "T'"], 输入串: '+ id * id $', 动作: match: 'id'
分析栈: ['$', "S'"], 输入串: '+ id * id $', 动作: T' -> ϵ
分析栈: ['$', "S'", 'T', '+'], 输入串: '+ id * id $', 动作: S' -> + T S'
分析栈: ['$', "S'", 'T'], 输入串: 'id * id $', 动作: match: '+'
分析栈: ['$', "S'", "T'", 'F'], 输入串: 'id * id $', 动作: T -> F T'
分析栈: ['$', "S'", "T'", 'id'], 输入串: 'id * id $', 动作: F -> id
分析栈: ['$', "S'", "T'"], 输入串: '* id $', 动作: match: 'id'
分析栈: ['$', "S'", "T'", 'F', '*'], 输入串: '* id $', 动作: T' -> * F T'
分析栈: ['$', "S'", "T'", 'F'], 输入串: 'id $', 动作: match: '*'
分析栈: ['$', "S'", "T'", 'id'], 输入串: 'id $', 动作: F -> id
分析栈: ['$', "S'", "T'"], 输入串: '$', 动作: match: 'id'
分析栈: ['$', "S'"], 输入串: '$', 动作: T' -> ϵ
分析栈: ['$'], 输入串: '$', 动作: S' -> ϵ
分析栈: [], 输入串: '', 动作: match: '$'
分析结果: True
请输入待分析串:程序结束
```

测试结果分析

从测试用例和结果来看，测试成功演示了文法分析器的整个工作过程，包括读取和处理文法，构建预测分析表，并解析输入串。以下是关键点的详细分析：

1. 文法读取与处理：

- 文法成功从输入读取并正确地显示了原始文法。
- 进行了左递归消除和左公共因子提取。

2. LL(1) 文法判断：

- 文法被成功判断为 LL(1) 文法，这意味着文法中的每个非终结符的所有产生式的选择集之间没有交集。

3. 预测分析表构建：

- 预测分析表被成功构建，表中为每个非终结符和输入符号组合正确地指定了相应的产生式。

4. 输入串解析：

- 输入串 "id + id * id" 被成功解析，分析过程中栈的操作和输入指针的移动正确地反映了文法的预测分析逻辑。

实验总结

收获

1. 理论与实践相结合：通过本次实验，我不仅复习并加深了对编译原理中上下文无关文法、左递归、左公共因子、FIRST 集和 FOLLOW 集的理论知识，同时通过编程实践将这些理论应用到具体的问题解决中，增强了理解。
2. 编程能力的提升：在实现文法分析器的过程中，我学习了如何设计并实现复杂的数据结构（如 Trie 树）和算法（如消除左递归、提取左公共因子、计算 FIRST 和 FOLLOW 集）。这不仅提升了我的编程技能，还加深了我对递归和迭代算法设计的理解。

3. 解决问题的能力：本实验中，我遇到并解决了多种编程问题，如算法优化、调试和错误处理。通过这些实际问题的解决，我学习到了如何更高效地使用编程工具和资源，如 Pytest 进行单元测试，确保程序的正确性和稳定性。

挑战

1. 文法的准确解析与处理：在实验中，确保文法正确解析并转换为适合处理的数据结构是一个挑战。特别是在实现文法转换算法（如消除左递归和提取左公共因子）时，需要确保转换后的文法依旧保持其语言的生成能力不变。
2. 算法的效率问题：随着文法规模的增加，算法的效率尤为关键。在实验中，我尝试优化算法以处理更大的文法，这包括优化递归调用、减少不必要的计算，以及合理使用数据结构来存储中间结果。
3. 用户交互和错误处理：设计用户友好的界面并进行有效的错误处理也是实验中的一个挑战。我需要确保程序能够处理不合规的输入，并给出有助于用户理解的错误信息